GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Calculating the Kullback-Leibler-Divergence on a Genome Scale

Tim van den Berg, Vincenz Dumann

MatrNr: —-

Supervisor: Marcus Vincent Boden, Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2022

Contents

List of Tables

List of Figures

List of Abbreviations

HPC High-Performance Computing

KLD The Kullback-Leibler-Divergence

PWM Position Weight Matrix

PCM Position Count Matrix

MPI Message Passing Interface

IO Input/Output

BPMN Business Process Model Notation

1 Introduction

This report is about the *"Practical Course on High-Performance Computing"*, held in the summer semester 2022 at the Georg-August-University Göttingen. Basic goal of the practical is to develop and benchmark a high-performance computing solution for a problem of choice.

1.1 Problem formulation

The problem on the basis of which this program was developed comes from bioinformatics: To find Transcription Factor Binding Sites, Position Weight Matrices of Transcription Factor Binding Motives can be used to calculate the Kullback Leibler Divergence. Due to the size of a human genome and the high number of Transcription Factors this can be a Big Data problem and needs enough computational performance to use an High-Performance Computing Cluster.

1.2 Motivation

At the beginning of the course, the course organisers presented a list of possible scientific fields for topics, including bioinformatics. Because of our biological background, we then enquired about topics in different bioinformatical work groups. The work group for medical bioinformatics of the UMG, lead by Prof. Beisbarth came back to us with the topic, that will be described in detail in the following chapters. The work group hopes to learn something about distributed computing from the newly developed solution to this well understood, classical problem. In addition, learning how to better utilize GWDG resources using distributed computing is a goal for the work group as currently most problems are computed on dedicated shared memory clusters hosted by the work group.

1.3 Organization of the report

This report is organized as follows: In chapter 2 some basics are introduced, which are essential for the further understanding of the report: Firstly, the Kullback-Leibler divergence, and secondly, some biological background information is presented.

Chapter 3 presents the project with a deeper level of detail and builds the bridge to the technical world: Here, solutions are presented to be able to work on the problem, and technical questions are clarified: What will a solution look like? How can it be parallelized? What improvements can be expected due to parallelization?

Chapter 4 describes the implementation of the questions from Chapter 3: Here, we present step-by-step how the solution was developed, from a very low performing, sequential, locally running implementation to a high performing server solution. This chapter also covers technical details and external libraries that were used.

Chapter 5 is all about performance: Here, the benchmarking strategy (including tools, hardware configurations etc.) are introduced and the performance of all imple-

mentations from chapter 4 are described and analysed. In addition, the scalability of all solutions is discussed.

Chapter 6 is about the challenges, their solutions and borders that occurred during the development process. In addition, some ideas on further optimizations are described, which were not actually done due to time limitations.

Chapter 7 is the conclusion of the report: The whole project is reviewed and learning achievements are discussed. It is decided on the basis of the goals if the project was successfully or not.

1.4 Authors

The authors of this report and responsible for its content are Tim van den Berg and Vincenz Dumann. Both are applied computer science students in the master's program of the University of Göttingen, the former in the 4th semester with a focus on high performance computing, the latter in the 3rd semester with a focus on software development. The division of the work is explained in more detail in the appendix.

2 Foundations

In this chapter, some basics are defined: First, some biological background is explained, in particular Transcription Factors are motivated. Then, the Kullback-Leibler divergence, is introduced.
Due to the limited size and scope of this report some simplifications are made, especially regarding the biological background. The presented information should suffice for the purpose of this practical course, however, when interested in the biological background, one should read a biology book.

2.1 Biological background: Genomes and Chromosomes

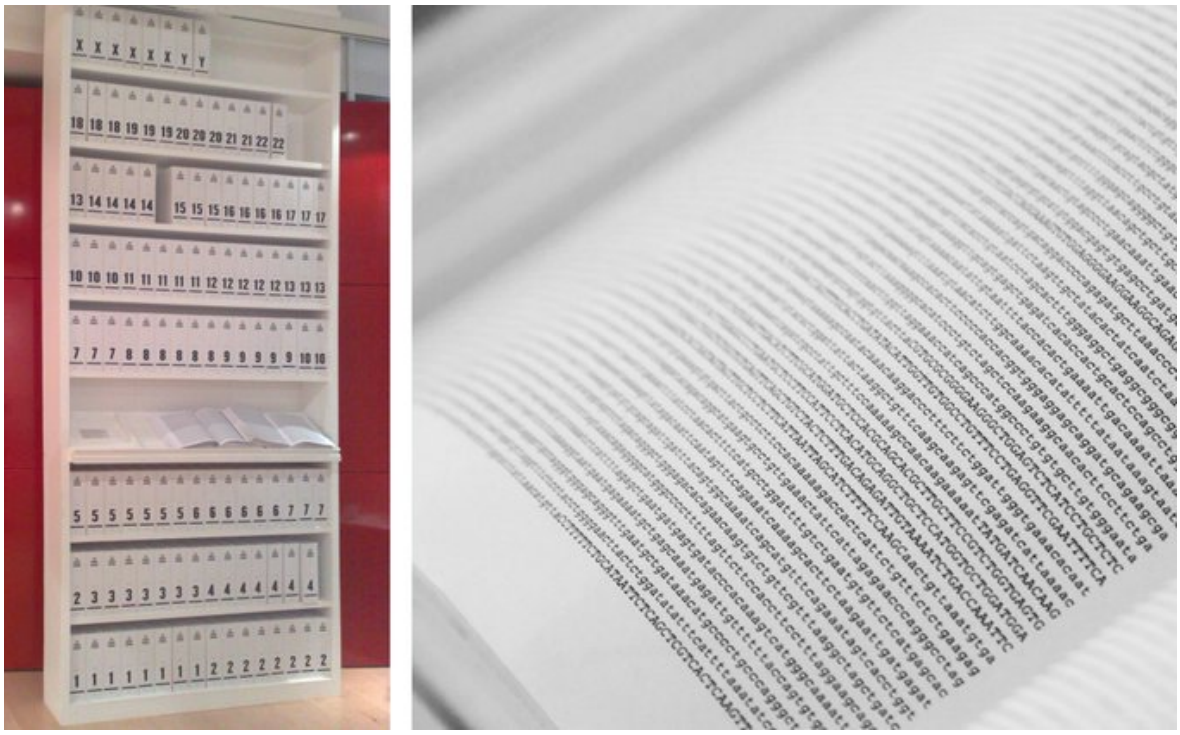

Figure 1: Human genome visualized as book series, images from [bro22]

In Figure 1 the human genome is visualized as a book series, exhibited in the 'Medicine Now' room (Wellcome Collection, London). This piece of art tries to transport the enormous size of the genome. The information of the 3,117,275,50 base pairs of the DNA was transferred into more than 100 book volumes, each 1000 pages, in the smallest readable possible font size (3px)[bro22].
The Genome is encoded in DNA and separated into 23 (24 when male) different chromosomes. These chromosomes differ in size. The largest chromosome is about five times larger than the smallest chromosome (chr. 1: 247 Mbp, chr. 21: 47 Mbp, see Figure 2).

All chromosomes exist twice per cell, as one copy comes from the father and one copy of the mother of a human. Therefore, there are 46 chromosomes in a human cell. The DNA is twisted a lot, when stretched out, every single cell in a human body would roughly contain 2 m of DNA. All DNA of all cells of a human combined would cover the distance from the Earth to the sun $\approx$ 1000 times. Combining the DNA of all humans would be longer than the diameter of the milky way.[Jüs]
The Genome contains the blueprints for products like Proteins and Enzymes that the cell needs to function. As humans develop over their life and cells have different tasks and a life cycle themselves, different cells need different amounts of different of these products. To produce a Protein or Enzyme, different steps take place, one can imagine a workflow. The first step is the so called Transcription. Here, a gene (the "blueprint") is read and a copy is produced, the so called messenger-RNA (mRNA).[Cam+16]

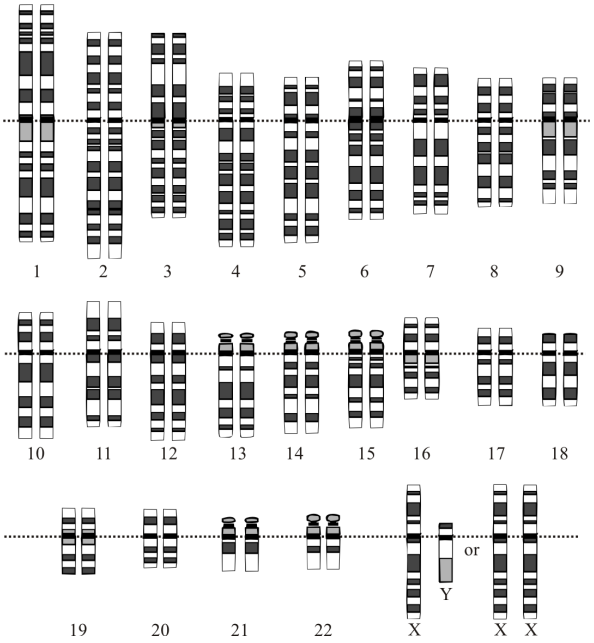

Figure 2: Human genome visualized by chromosome sizes. Chromosome 1 is more then six times bigger then chromosome 22. Image from [wik22]

To influence which genes are transcribed, so called Transcription Factors can enhance or suppress the Transcription of specific genes. Transcription Factors bind on the DNA. It is important to understand where certain Transcription Factors bind in order to understand what they do and what the respective genes do. One possibility is that Transcription Factors bind at certain positions because there are certain motives in the order of the bases on the DNA. To find these Transcription Factor Binding Sites, one can use the Method used here. However, just running our method will probably not give very accurate data, normally one has to combine this with other methods, but the goal here is not to gain biological insights.

2.2 Mathematical background: The Kullback-Leibler-Divergence

The Kullback-Leibler divergence[S K51] (KLD) is a measure of the difference between two probability distributions. Typically, one of the distributions represents empirical observations or a precise probability distribution, while the other represents a model or an approximation. Formally, the Kullback-Leibler-Divergence for the probability functions P and Q of discrete values can be determined as follows:

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log(\frac{P(x)}{Q(x)}) \tag{1}$$

From an information-theoretic point of view, the Kullback-Leibler divergence indicates how much space per character is wasted on average when an encoding based on Q is applied to an information source that follows the actual distribution P.

2.2.1 Using the Kullback-Leibler-Divergence on Genomes

- x is a base from X={A,C,G,T}
- P(x) is the probability of base x at the respective index in the PWM/Transcription Factor Binding Site (PWM[x, index])
- Q(x) is the background probability of base x in the genome (the percentage of x in the genome)
- The Kullback-Leibler-Divergence gives a score that is higher if the probability for a Transctiption Factor Binding Site is higher at that position
- in genomic contexts a logarithm base 2 is used
- Note that the Kullback-Leibler-Divergence depends on the length of the PWM

2.3 Software Development Background: Design Pattern

Design patterns are proven solution templates for recurring design problems in architecture as well as in software architecture and development. They thus represent a reusable template for problem solving that can be used in a specific context. Originally, 23 design patterns were described, which were divided into three categories[Gam97]:

- Creational Patterns
  - Concern object creation. They decouple the construction of an object from its representation. The object creation is encapsulated and outsourced to keep the context of the object creation independent of the concrete implementation.
  - Well known examples are Factory Pattern and Singleton Pattern.
- Structural Pattern
  - Concern relationships between classes. They provide ready-made templates for common challenges.
  - Well known examples are Facade Pattern and Composite Pattern, which will be described in this chapter.
- Behavioral Pattern
  - Concern behavior of the Software. They are used to increaste the flexiblity of software.
  - Well known examples are Chain-of-responsibility and State Pattern, which will be described in this chapter.

2.3.1 State Pattern

The State Pattern provides that the states of objects are also objectified and designed as implementations of a common superclass. State-specific behavior can thus be implemented in the corresponding state classes, which greatly facilitates the extension of the software. For example, in modeling a door, the current state would be split into two classes ("open" and "closed"), each inheriting from the abstract class "door state". The function for opening the door would then be implemented concretely in the class "shot". This implementation facilitates the extension (e.g. adding the states "ajar" and "wide open" as further states of "open") compared to an implementation via boolean attributes.

2.3.2 Composite Pattern

The composite pattern is applied to represent part-whole hierarchies by combining objects into tree structures. The basic idea of the composite pattern is to represent both primitive objects and their containers in an abstract class. Thus, both individual objects and their compositions can be treated in a uniform manner[Gam97].

3 Problem Analysis

After the introduction and the basics, this chapter - initially independent of technical limitations - analyzes the problem at hand in detail. Here, the requirements for the solution are established as well as various possibilities are examined as to how these can be achieved. In addition, the question of how far parallel executions are possible will be addressed, as well as hypotheses on how these will affect performance. Finally, metrics will be defined that will serve as indicators of success or failure.

3.1 Resources

Two resources are needed for the implementation: One is the Position Weight Matrices of the Transcription Factor binding motives, the other is the complete Human Genome. The Position Weight Matrices of the Transcription Factor binding motives can be downloaded as Position Count Matrices (aka Position Frequency Matrices) from the JASPAR database[JA+] as a text file with about 10,000 lines, including 1956 matrices at a size of ≈750 kilobytes. JASPAR is the leading database in the field and describes itself in the following way:

> "JASPAR is a regularly maintained open-access database storing manually curated transcription factors (TF) binding profiles as position frequency matrices (PFMs). PFMs summarize occurrences of each nucleotide at each position in a set of observed TF-DNA interactions."[JA+]

The sequence of the human genome can be downloaded from the U.S. National Library of Medicine website[NCf13], which belongs to the National Center for Biotechnology of

the United States of America. The resulting file is a Fasta formated text file of almost 3.5 gigabytes.
For a better understanding, both resources will now be presented by means of an example. One example Position Count Matrix of the downloaded Data Set is the following:

```
>MA0004.1 Arnt
A  [     4     19      0      0      0      0 ]
C  [    16      0     20      0      0      0 ]
G  [     0      1      0     20      0     20 ]
T  [     0      0      0      0     20      0 ]
```

One entry consists of one row starting with a ">" containing meta information (here: the matrix ID MA0004.1 and the Transcription Factor name Arnt) and four rows representing the four different bases Adenine, Cytosine, Guanine and Thymine. Each column in a matrix should sum up to the same number as it contains the number of occurrences of a base at a specific location in an observed Transcription Factor - DNA binding. The Position Count matrices are processed further, as will be shown in chapter 3.2.1.

The human genome, divided into 46 chromosomes, is in its decoded form a combination of the bases A, T, G, and C, each in normal and capital letters:

```
GTCTTTTCATTGCACACAAATTGAACTTTTAAAAGAGGTGCAAATGTCCTGTA
ATacggtttgtctgtgtccccacccaactcgcaccttgaattgtagtttccat
ggtgaagataattgaatcatggggccagttcCCCCCATCCTGTTCTCCTAATA
gttttataaggggcttccccttttggcggggctctcattcttctctctcctgag
cccttccgccacgattgtaagttttctgagacttcccagccctgcagaactgt
aattacccattcttgggtatatctttattggcagtgtgagagcagactaatat
```

Lowercase letters indicate that the corresponding bases are not unique, while uppercase letters indicate clear positions. Each person has a different genome, some positions are the same in all people, some define the differences between us. Some positions are not clear at all, those are marked with an 'N', they appear mostly at the edges of a Chromosome in the so called Telomeres which protect the chromosomes.

Now that the input data and the goal of the project are clear, the algorithm that performs the corresponding transformation must be designed. The following subchapter is dedicated to this task and provides an overview of the different approaches used during this project to solve the problem.

3.2 Implementation drafts

In this section, plans for implementation are elaborated, and the process from a sequential, simple solution to a parallel and thus much more complex one is shown. Here, technical limits are only observed to a limited extent; these only come into play in the following chapter. The general workflow is shown in Figure 3 and will be presented in more detail in the following.

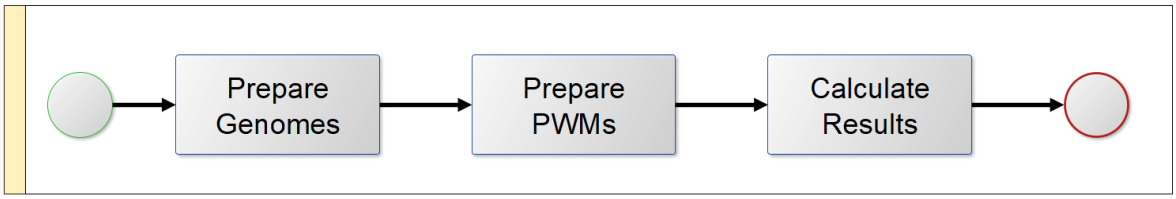

Figure 3: Basic sequential execution of the calculation. Basic workflow of the program, simplified into the three main activities.

3.2.1 Preparation of Data

Before any calculation can take place, the used raw data, described in the former chapter, needs to be prepared.

Prepare Genomes The downloaded file containing the genome is not yet ready for direct processing, for several reasons:

- The file could be corrupted: With several gigabytes of text file, it is not impossible that letters other than 'A', 'T', 'G', 'C' or 'N' (upper and lower case, respectively) have inadvertently entered the file. This should not happen, to be on the safe side (and because one has to iterate over the file anyway), all other letters are replaced by 'N'.
- The file contains upper and lower case letters. However, the information encoded by this, whether they are fixed at the corresponding positions, is irrelevant for this application, but requires additional comparison operations.
- In these comparison operations, the use of letters is counterproductive. An integer-level comparison promises a significant improvement for little effort. Therefore, all numbers are transformed to the smallest format available - 8-bit integers, which in turn are loaded into a numpy array. In the second parallel approach, which is described in detail below, the file is also split on chromosome level. In this case, a Numpy array is created for each chromosome, and these are then pickled.

Prepare Position Weight Matrices In order to calculate the Kullback Leibler Divergence, the downloaded Position Count Matrices must be converted to Position Weight Matrices. For each column, the relative weight of each base is calculated: If a column consists of the four values [5, 10, 5, 0], this is transformed to [0.25, 0.5, 0.25, 0]. This operation is applied to all matrices. The example shown before looks now like this:

```
>MA0004.1 Arnt
A  [  0.25   0.95      0      0      0      0 ]
C  [  0.75      0      1      0      0      0 ]
G  [     0   0.05      0      1      0      1 ]
T  [     0      0      0      0      1      0 ]
```

In order to avoid taking the logarithm of zero when calculating the Kullback Leibler Divergence, a so called pseudo count is added. This is a very small number, that is added to every position in the Position Weight Matrix.

3.2.2 Sequential implementation

The implementation of a sequentially running program using the formula is trivial. Using the hints from section 2.2.1, the calculation can be performed without further effort.

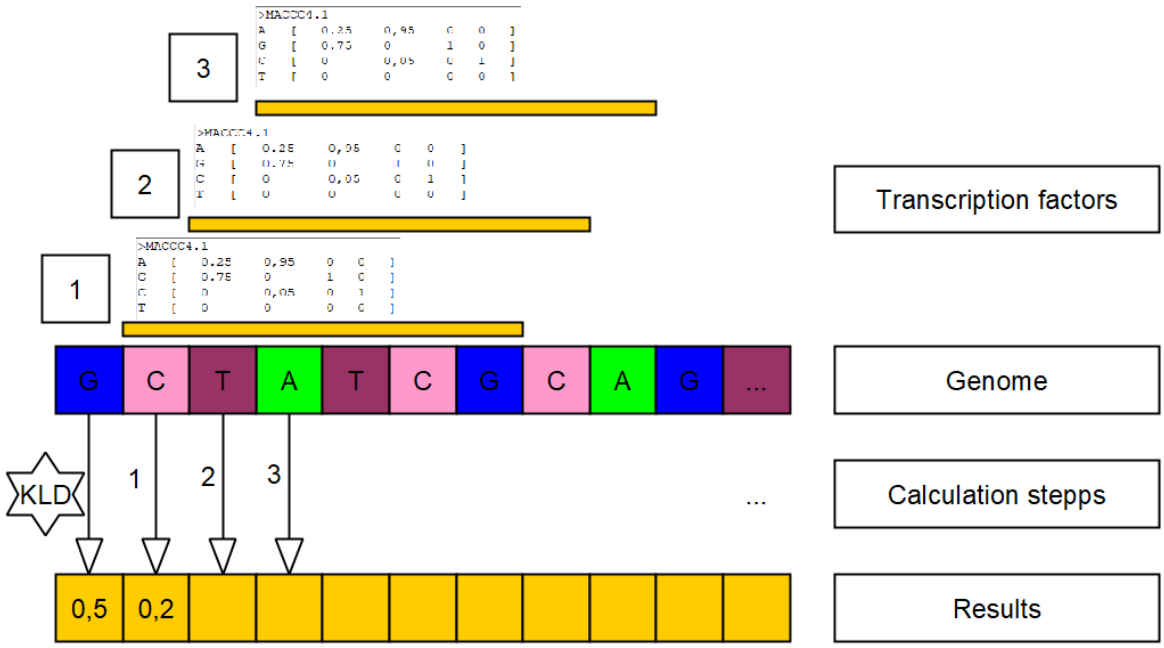

Figure 4: Algorithm sketch. The Position Weight Matrix (yellow bar) slides over the genome and at each position the Kullback Leibler Divergence is calculated. The results are stored in an array.

Figure 4 shows the basic idea of the algorithm: The Position Weight Matrix (PWM) is applied at every position of the Genome. Every time, the Kullback Leibler Divergence is calculated using the information from chapter 2.2.1.

A sequential solution in (python-like) pseudocode for the Kullback Leibler Divergence (formula 1) can look like this:

```
genome = "ATGCGTG...."
pwm = [[...][...][...][...]] # rows are bases
background = [0.25, 0.25, 0.25, 0.25] # background probabilities
kld = [] # empty result array
for index_genome in 0 : genome.length:
    for index_pwm in 0 : pwm.length:
        index = index_genome + index_pwm
        current_base = genome[index_genome]
        p_base = pwm[current_base, index]
        q_base = background[base]
        kld[index_genome] += p_base * log2(p_base/q_base)
```

3.2.3 Runtime Predictions

In the pseudocode shown in the previous chapter, it is obvious, that one has to loop over the genome and the PWM. Therefore, an quadratic upper bound of $\mathcal{O}(\text{genome.length} \times pwm.length)$ can be assumed.
Now it would of course be interesting how fast the sequential algorithm shown in pseudocode can actually run on a computer. First, let's calculate the number of iterations, that have to be performed. The genome is 3,117,275,501 bases long. The PWMs vary in length between 5 and 35. In total, the 1956 PWMs have a length of 24391. This results in 76,032,871,824,010 iterations in total. A processor works in the GHz dimensions. Assuming one iteration takes one clock cycle the calculation takes ≈21 hours. However, a CPU is normally a bit faster than 1 GHz. Nevertheless, an iteration should take much longer than one clock cycle as there is a division included and probably some calls to memory as the data does not fit in the cache. A realistic lower bound would therefore be a few days to calculate everything.

At each iteration one floating point number (the Kullback Leibler Divergence) is created and has to be stored. If one uses ASCII to encode the resulting text files and uses 10 characters to store one value this results in 10 Byte per KLD value and to 760 TB of data. A modern SATA SSD can write at a speed of roughly 500 MB/s, here it would take ≈18 days to write this much data to a single SSD. Using a HHD (150 MB/s) it would take ≈59 days. Therefore, if one would really would want to calculate the Kulback-Leibler-Divergence on the human genome for all Transcription Factors, one should think of a smart way of storing the results. One possibility to remove a lot of unneeded data from the results would be to use a threshold and just store the results with a Kullback-Leibler-Divergence higher than the threshold.
Reading in the 3 GB genome text file does not take a significant amount of time when compared with the writing times calculated above.

3.2.4 Implementation Goals

The overall goal of the task described in this report is to optimize the performance of this calculation as much as possible. However, a few more subordinate features are also to be implemented in order to obtain a complete, server-based solution. These goals, sorted by importance, are defined as follows:

1. Overall goal: High-performance, concurrent implementation for calculating the Kullback Leibler Divergence of a Transcription Factor Binding Site.
   (a) Without further knowledge of performance, we set a goal of completing the full calculation in under 1 hour.
   (b) To create the baseline for the performance increase, a sequential implementation must first be created.
2. Creation of a worker infrastructure: A main process should be able to start and manage various worker processes, which perform the actual calculations in parallel.
   (a) The infrastructure should be fail-safe, which means that the calculation continues even if a worker process fails. The main process should reassign the failed workers tasks.
   (b) A heartbeat model should be implemented to detect failed workers.
   (c) The failure of workers should be simulatable.

So the next step is obvious, but far from trivial: parallelizing the sequential solution.

3.2.5 Parallelization

The first question that arises when one wants to parallelize such software is whether to implement parallelization at the task or data level. Data parallelism means concurrent execution of the same task on each computing core, whereas task parallelism means concurrent execution of the different tasks on multiple computing cores.

Starting with task parallelization, it should be noted that there are 3 tasks, two of which are independent of each other (the preparation of the transcription factors as well as the preparation of the genomic sequence data), while the actual calculation has to wait for the completion of both of them. Therefore, the data preparations can be done parallel. This approach has two major drawbacks:

1. Data preparation is expected to be a relatively small percentage of the total processing time. The main part will be the actual computation of the Kullback-Leibler-Divergence. So the expected overall speedup here is relatively small.
2. This approach is not scalable: There are exactly two processes that run in parallel and are not further split. The only possibility for faster processing (assuming an optimal implementation) is therefore better hardware.

Of course, this small expected performance gain can also be taken, but the focus should be on parallelizing the actual calculation as well as possible in a scalable manner. In this case, it is therefore a logical choice to split the data and process it in parallel. The data can be separated on genome level as well as on transcription factor level. In the following, only the separation on the basis of the transcription factors will be discussed, since the implementation on the genome level is the corresponding.

Figure 3.2.5 shows the algorithm with the use of 2 Workers. However, this number is not fixed - in contrast to the task-parallel approach - and can be increased as desired. The implementation provides that the main process first generates work packages, consisting each of a PWM and a chromosome. The worker processes report to the main process when they are idle and are assigned work packages, which they process. When the worker is done, the results are reported back. Afterwards the worker is idle again and a new work package can be assigned.

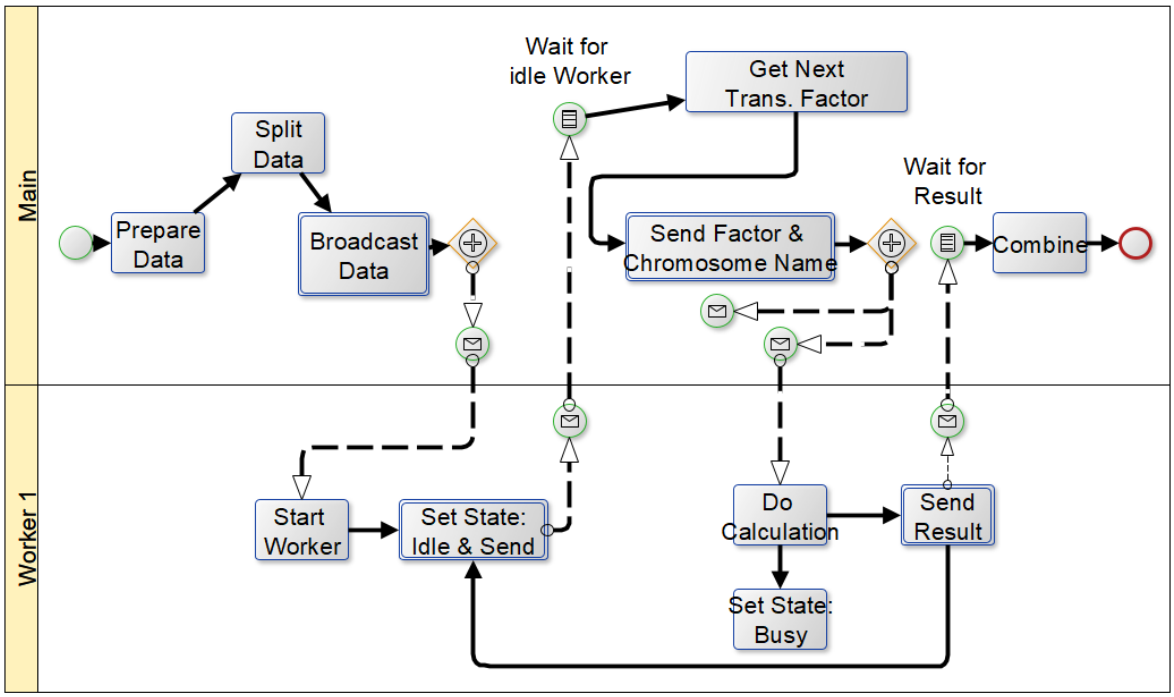

Figure 5: Data parallel execution of the calculation, here displayed with only one worker. The Amount of workers is flexible, all displayed messages are sent to all of them. The high number of communication points in this diagram highlights the focus of this approach.

With the creation of such a concrete plan, the design phase can be considered complete. The next chapter will now deal with the implementation, i.e. the technical realization.

4 Implementation

In this chapter, the individual implementations are presented in detail. In each case, the things that were not implemented as in the drafts from the previous chapter are discussed, as well as technical peculiarities and difficulties that occurred. In addition, the Wall Clock Times, which are measured with the runs, are indicated in each case. The following chapter is dedicated to these measurements in detail and analyzes the performance.

4.1 General Implementation

In general, Python 3.9.0 was used. Furthermore, numpy 1.21.3, matplotlib 3.4.3, numba 0.56.0 and mpi4py 3.1.2 was used.

A special feature of the implementation presented here is the orientation towards a modern, object-oriented programming style. A detailed discussion of the advantages and disadvantages of such an approach is far beyond the scope of this report, but in this educational project the deciding factor for choosing this approach was to learn and evaluate this approach in a practical, yet self-contained environment. A short conclusion on this can also be found in the review of this project at the end of this report.

The full source code can be found at `https://gitlab-ce.gwdg.de/tim.vandenberg/practical_hpc`. At this point, we want to pointed out that the complete commit history is available to any interested reader. This report will therefore only go into more detail about the actual technical implementation if there were significant deviations from the draft during implementation, or if very specific details need to be pointed out. Furthermore, in order not to go beyond the scope of this report, the comments in the source code are explicitly referred to for a deeper understanding.

4.2 Sequential implementation

The sequential implementation was completed without significant problems. There are no significant deviations from the analysis or the draft detailed in the previous chapter. To follow clean code principles, helper functions were used and refactored into a utils class, which now provides basic functions that are not directly related to the corresponding approach, but are nevertheless needed by one or more approaches. Type hints comments and docstrings were used to make the code better understandable and readable.
The pseudocode from chapter 3.2.2 was used as a guideline during the development. Therefore, the function calculating the The Kullback-Leibler-Divergence (KLD) is fairly similar:

```
1 def calculate_kld(sequence: np.ndarray, pwm: np.ndarray,
2                   background: np.ndarray) -> np.ndarray:
3     results = np.zeros(len(sequence) - pwm.shape[1] + 1)
4     for start_idx in range(len(sequence) - pwm.shape[1]): # loop sequence
```

```
        kld = 0.0
        for cur_pwm_pos in range(pwm.shape[1]): # loop pwm
            b = sequence[start_idx + cur_pwm_pos] # nucleobase at index
            if b == 4: # no specific base
                # equal to 0, do nothing
                continue
            kld += pwm[b,cur_pwm_pos]*np.log2(pwm[b,cur_pwm_pos]/background[b])
        results[start_idx] = kld
    return results
```

4.2.1 Execution Results

The most primitive approach is of course the non-optimized, sequential approach. This does not mean, that this version was as slow as possible as numpy operations we managed to use numpy. Nevertheless, many iterations have to be performed using relatively slow python for loops. For a runtime prediction see chapter 3.2.3. This first run yields an interesting baseline against which further measurements can be compared. The whole genome and one PWM of length 6 was used, the run was executed on the local machine (AMD Ryzen 5700g, an SSD and 16GB of Memory).

Table 1 shows the run times of the different processing steps.

Task	Name	Time (in h)	Percentage (in %)
Wall Clock Time	T_{total}	07:58:31	100.00
Calculations	T_{calc}	07:52:20	98.71
Reading to Disk	T_{read}	00:03:14	0.78
Writing to Disk	T_{write}	00:02:55	0.61
Miscellaneous	T_{misc}	00:00:02	0.00

Table 1: Unoptimized Sequential runtimes on local machine, featuring AMD Ryzen 5700g, an SSD and 16 GB of Memory

The run time of ≈8 hours for a PWM with a length of 6 can serve as a baseline. If one assumes a linear growth of the run time based on the PWM length, the total computation time for all PWMs (total length: 24391) over the complete genome is about 3 years and 8 months. This is of course not feasible in any circumstance and should make clear why optimizations and parallelization are important. The measured runtime is many orders of magnitude larger, than the runtime predictions in chapter 3.2.3 suggest as a lower runtime border.

The distribution of times, see also table 1, shows very clearly where the bulk of the run time was spent: on the computation itself, not on reading or writing the data. Thus, this is the obvious starting point for the start of the optimization.
Another problem with this implementation was the amount of RAM used. More then 20GB were in use during the execution. Therefore, the computer started swapping, which could have caused an even worse performance.

4.3 Improvements with Numba

The most important improvement to the sequential approach is the introduction of Numba. The creators of Numba describe their library as follows: "Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN"[num22]. Basically, this means that Numba does compile an appropriately marked block of source code to optimized machine code using LLVM instead of using the (much slower) python interpreter.

Adding Numba in a python-based solution is simple, if not trivial: the corresponding function is simply provided with a decorator in which the appropriate settings for the affected code block are defined. Figure **??** shows this "implementation" - whereas such a trivial change hardly deserves the name "implementation".

```
1  from numba import jit
2  @jit(nopython=True, parallel=False)
3  def calculate_kld(sequence: np.ndarray, pwm: np.ndarray,
4                    background: np.ndarray) -> np.ndarray:
5      results = np.zeros(len(sequence) - pwm.shape[1] + 1)
6      for start_idx in range(len(sequence) - pwm.shape[1]): # loop sequence
7          kld = 0.0
8          for cur_pwm_pos in range(pwm.shape[1]): # loop pwm
9              b = sequence[start_idx + cur_pwm_pos] # nucleobase at index
10             if b == 4: # no specific base
11                 # equal to 0, do nothing
12                 continue
13             kld += pwm[b,cur_pwm_pos]*np.log2(pwm[b,cur_pwm_pos]/background[b])
14         results[start_idx] = kld
15     return results
```

The only changes made to the code are shown in the first two lines. The effects of this small change are impressive. The run times can be seen in the next section.

4.3.1 Execution Results

The implementation optimized by numba was executed with the same PWM over the complete genome. For better comparability, the same computer was used as for the original sequential execution. The run times, as well as their distribution to the individual program parts can be taken from table 2.

Task	Name	Time	Percentage
Wall Clock Time	T_{total}	00:08:09	100.00
Calculations	T_{calc}	00:01:59	23.72
Reading from Disk	T_{read}	00:03:12	39.26
Writing to Disk	T_{write}	00:02:58	36.40
Miscellaneous	T_{misc}	00:00:03	0.61

Table 2: Sequential runtimes on local computer.

As one can see in table 2, the time for the calculation part has gone down massively - 8 hours became 8 minutes only by using Numba. Since this acceleration only affects the calculation part, the share of the corresponding time has shrunk massively from 98.71% to 23.72%.

Memory consumption, on the other hand, remained high. This was to be expected, since Numba only affects it minimally.
However, this practical course is primarily not about the optimization of sequential code, but about concurrent computing. These are - as soon as implemented - also not executed on the local computers, but are supposed to run on GWDG servers. In order to be able to use the implementation with Numba as a baseline measurement, the optimized sequential program was executed on the server (amp014). The measurements can be found in Table 3.

Task	Name	Time	Percentage
Wall Clock Time	T_{total}	00:14:26	100.00
Calculations	T_{calc}	00:04:07	28.52
Reading from Disk	T_{read}	00:07:02	51.03
Writing to Disk	T_{write}	00:02:55	20.21
Miscellaneous	T_{misc}	00:00:02	0.23

Table 3: Sequential run times on the server(amp014).

As one can see, the execution time is significantly longer, ergo the performance on the server is lower. Especially the reading part (from 3 to 7 minutes) as well as the calculations (from 2 to 4 minutes) had a big influence on this increase. In particular, the increase in read access was to be expected, since the server does not have SSDs directly attached to the execution unit.

Of course, it would be possible to optimize the sequential share even further using numba, but this implementation should now lay the foundation for the parallel implementation - and the measurements represent the baseline value accordingly.

4.4 Implementation Parallel Solution

The implementation of this approach could generally be implemented as intended by the design, see Figure 3.2.5. At this point, however, the implementation of the communication between main and worker process should be discussed, since this is the decisive technical detail that is relevant for this approach.

4.4.1 Implementation of Communication with MPI

The Message Passing Interface (MPI) is a standard, that enables processes to communicate with each other. The python implementation mpi4py uses the pickle library to serialize data and sends them. Numpy objects can be send directly, therefore this is much faster and was used wherever possible. The syntax of a blocking point-to-point

communication is demonstrated in the following example.

```
import numpy as np      # import numpy
from mpi4py import MPI  # import MPI library

comm = MPI.COMM_WORLD        # initialize communicator
rank: int = comm.Get_rank()  # get rank

if rank == 0:  # main process (sending in our case)
    # send numpy (np) array
    send_array: np.ndarray = np.zeros(5) # array we want to send
    comm.Send(send_array, dest=1, tag = 1) # capitalized Send

    # send other data
    send_data = {'a':7, 'y':42}
    comm.send(send_data, dest=1, tag=2)

elif rank == 1:  # receiving process
    # receive numpy (np) array
    receive_array: np.ndarray = np.empty(5, dtype=np.float64)
    comm.Recv(receive_array, source=0, tag=1)

    # receive other data
    receive_data = comm.recv(source=1, tag=2)
```

One can see, that the Recv() function (for numpy objects) writes directly into an initialized numpy object, while the normal recv() function returns an object.
This blocking point-to-point communication was used for communication between the main process and the worker processes. Furthermore, a broadcasting operation was used to broadcast the background probabilities and the genome data from the main process to the workers. When a worker was done with the calculation and ready to transmit the results back to the main process, an empty message is send to the main process. The main process loops through all worker processes constantly and checks with the iProbe() function if a respective message was send. If it was send, the main process interacts with the worker to get the results and provide the worker with a new task.

4.4.2 Challenges during Implementation

During the work on this parallel implementation we learned that debugging parallel, communicating code is not trivial due to several reasons.
The implementation of the MPI standard by the mpi4py library is not very intuitive to use, as the usage of numpy object passing and normal message passing differ substantially. The numpy versions of the MPI commands are capitalized and one would assume that the behaviour is similar to the normal MPI commands. However, while

the normal commands return the received object, the numpy version writes into a initialized numpy object given as an argument. The resulting error message when doing anything wrong was cryptic to us.
Furthermore, next time we would link our IDE with the remote server. It was very draining to always commit, push and pull just to make a small change to the code. Also, testing changes in the code took longer as we sometimes had to wait for our slurm job to start.
During the development of this application, confusion arose again and again as to which Id is assigned to which communication channel. In response, the code was refactored and the Ids were organized into so-called "enums". An enum, short for enumeration, is a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over[pyt22]. The main advantage of enums is that they are much easier to read and thus provide better clarity for the developer - and thus not an advantage of a technical nature. Instead, the use of meaningful, carefully chosen enums reduces the risk of bugs whose origin lies in human error. The use of enums follows the zen of Python principle "Readability counts."

4.4.3 Execution Results

The corresponding execution times with different numbers of worker processes, are listed in Table 4. All nodes were amp*** nodes which are Cascade Lake Intel Platinum 9242 machines[GWD].

#Worker	Time	Time/Worker (s)	Speedup	Speedup/Worker
1	39:12	2352	1	1
4	14:05	213	2.75	0.68
8	7:13	54	5.4	0.67

Table 4: Parallel 1 runtimes on Server

When looking at these values superficially (for a more detailed analysis, see Chapter 7), it is immediately noticeable that the run with one worker (≈39 minutes) took significantly longer than the execution of the sequential solution (≈8 minutes). However, this duration is already significantly undercut with 8 workers. Furthermore, it can be seen that while quadrupling the workers from one to four could only achieve a speedup of 2.5, doubling again could achieve an almost linear decrease in processing time.

Due to the still very long processing time for a run with one PWM, further tests with multiple PWMs were not performed. It is obvious that the approach is not optimal, since the main goal here was to implement a working MPI-based solution and not yet to achieve large performance leaps. This is the focus of the second approach, the implementation of which is the subject of the next chapter.

5 Improved Parallel Solution

Another approach would be to first divide the data by the number of free workers and give these packets to the workers. This has the advantage that much less communication would be necessary, but the work packets would be much larger. A failure of a worker would mean that a large packet would have to be split again later, increasing the complexity of the algorithm. It would also be less flexible to respond to a fluctuating number of workers, making this solution less scalable.

The second approach, implemented as part of this project, is intended to compensate for some of the drawbacks of the first and provide better performance. To achieve this, the following points should be considered:

- The implementation should contain less communication overall - both quantitatively and qualitatively. While in the first approach a lot of messages containing large amounts of data (complete chromosomes) were sent back and forth between the main and the workers, in this second approach only a few, small amounts of data should be sent if possible. Of course, the workers still have to receive chromosome data, and the smallest possible amount of data that is sent in total corresponds to the length of the chromosome, but great savings can be expected here. Depending on the hardware configuration, it could be better to let the worker processes read the data from disk themselves.
- Furthermore, the remaining bottlenecks should be eliminated - especially the writing of the results. For this, the workers should be enabled to save their respective results in files themselves, instead of sending them back to the main, which then takes over this task - this would also save further communication.
- If possible, although not the main goal, the used memory should be reduced significantly. Of course, the performance is in the foreground, but it will always suffer when a system has to swap memory.

The implementation of a software solution that meets these requirements is shown schematically in Figure 6 (exemplary with one worker, of course this can be almost any number): As before, the data is first prepared. However, the genomic sequence is split into different files to be able to outsource all Input/Output (IO) work to the worker processes. Then tasks are generated, but this time they are distributed differently. Tasks are assigned according to the data that is needed. If worker 1 started on genomic sequence chunk 1, it will get tasks regarding that genomic sequence chunk until there are no more tasks regarding it.

5.1 Implementation Parallel Approach 2

The implementation of this new version worked without major issues. However due to the reasons described in chapter 4.4.2, the implementation took quite a while and was

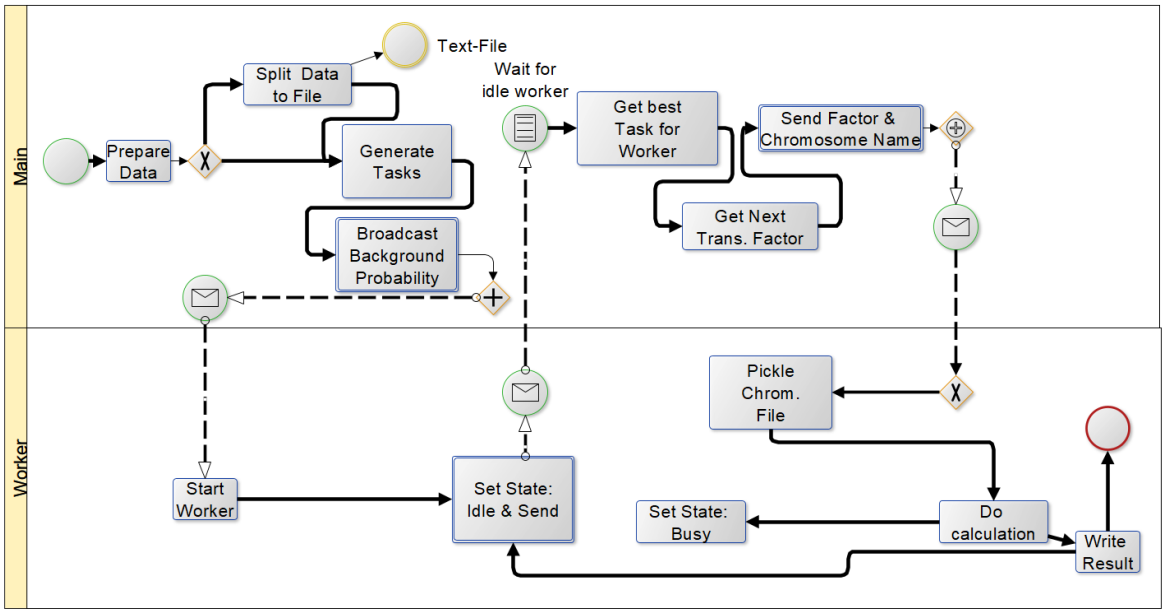

Figure 6: Optimized data parallel execution of the calculation. The amount of communication was reduced drastically and the preparation of data extended.

quite challenging mentally. In the end, this lead to a lot of time pressure and some features and performance analytics could not be implemented or performed. This will be described in detail in chapter 6 and chapter 7.
The requirements and Business Process Model Notation (BPMN) graph described above and shown in Figure 3.2.5 were the foundation for this implementation.

5.1.1 Improved Data Preparation

In contrast to the previous solution, here the data preparation and storage of the results is performed as much as possible by the worker processes instead of the main process. This enables the main process to handle large amounts of worker processes as the blocking communication (receiving GBs of results) does not interfere with the process management duties of the main process. However, it is not trivial to prepare data in parallel. To avoid file access of all workers on the same file, the main process can now split the genome into separate files, each containing one chromosome. Here, solution with a custom chunk size as described before would be even more ideal, but was not implemented due to time reasons.
The rest of the data preparation happens in the worker processes. If a worker process gets a task from the main process, consisting of a chromosome name and a PWM, the worker process first checks if the respective chromosome data is already in its memory. If this is not the case, it checks, if a pickled version of the preprocessed data is available. If one is available, the process reads the data and starts with the computation. If not, the process reads the chromosome sequence text file, that was created by the main process earlier and does the same data preparation steps as described in the previous chapter. The resulting numpy array is then pickled as another process may want to

use it later.
Each chromosome file should just be prepared once, if the number of workers is smaller than the number of chromosomes. To optimize this further, one should make sure that each chromosome is preprocessed by exactly one worker, this would not be hard to do, but was omitted due to time reasons. The results are pickled in the end, by the worker process if this is wanted. Due to the large output, the pickleing and storage of the output files was omitted for the benchmarks, as with the previous runs.

5.1.2 Communication with MPI

The communication between the main process and the worker processes is very minimal in this approach, as no results or genomic sequences are transmitted. The only broadcasted data are the background probabilities (numpy array with length 5). Otherwise, the worker processes send their idle state and receive tasks, consisting of a PWM (numpy array) and a chromosome name (string). Finally, there is a "keep working" boolean value, that is send to every worker while there is still work to be done. This was a first step towards a heartbeat model, as a worker could have get a new task even if there were no tasks left in the meantime. However, this feature was not implemented due to time reasons, see chapter 6.

5.1.3 Execution Results

Unlike the previous implementation, the best possible performance is aimed for here. Therefore, more extensive tests were carried out here and not only the number of workers varied more, but also the number of PWMs processed. The values, as well as the respective relative acceleration, can be taken from the table 5. In the test runs listed here, a PWM with a length of 6 was used in the first run, and 4 PWMs with different lengths and a total length of 36 - which is the relevant quantity at this point - were used in the second run.

#Worker	len(PWMs)	Time	Time/len(PWM) (s)
1	6	01:05:15	652.50
1	36	03:32:02	767.03
8	6	00:10:15	102.50
9	36	00:38:03	63.42
10	6	00:09:00	90.00
12	36	00:26:45	44.58
16	6	00:08:24	84.00
16	36	00:28:31	47.53
21	6	00:06:59	69.83

Table 5: Parallel Approach 2: Wall clock times

6 Removed Features

In the course of implementing the three approaches (especially the third approach), there were some features that were initially included, but did not make it into the final builds. This had different reasons, and the features, and the corresponding backgrounds of non-observance, will be presented in detail in this chapter.

Basically, a slightly different approach was planned for the second approach: The implementation was to generally have the basic features of a scheduler - Different workers were to be able to sign up, then be monitored, be assigned tasks, and be supervised. However, as the project progressed, the focus shifted further and further away from this approach, and more and more attention was paid to optimizing the actual performance. This in turn had several reasons:

- The simplest, but most important reason: Time. Implementing the best possible performance was much more time-consuming than originally planned, which meant that these rather secondary features were cut out accordingly.
- The more features are implemented around the actual task, the more difficult it becomes for colleagues from bioinformatics to understand the code - especially since they are predominantly not trained computer scientists, but people whose focus is on the biological part of bioinformatics. For them, the goal is to implement a solution that is as easy to understand as possible, focusing on the essentials. The importance of this educational approach to the project, however, only became clearer over time - a change in the priorities of a project inevitably ensures an adjustment of the requirements.

Now that the different reasons have been discussed, these features will be presented individually. In each case, it is also indicated how far the corresponding feature has already been implemented and how far approaches can still be found in the final version. Overall, the analysis, design, and started implementations for these features took about one-third of the total processing time, so they were a non-negligible factor in the implementation.

6.1 Registration/Deregistration of Workers

In the first versions of the second approach, it was intended that workers could register with the main process via an interface, either individually or in a group, in order to then be provided with work by this process. For this purpose a register(), as well as a deregister() function for workers or worker groups was implemented, over which the appropriate funtionality was made available. The registerWorker() function, unlike the registerWorkerGroup() function, was already completed, and a relic of this implementation can still be found in the architecture of the code: The management was supposed to be done by a state pattern[Gam97], of which always the states "Busy" and "Idle" are left as objectified states. These have in the final version only a rather subordinate role as parameters with the inquiry for work of a worker at the Main, formed however originally the foundation of a condition-based control.

The group registration of workers was not implemented, but was only planned as an extension of the simple worker registration and therefore trivial: While for registerWorker() a worker object was planned as parameter (in the early versions this contained only a worker ID, for this see next subchapter), for the group registration a function was planned, which contained an object, which contained beside a group ID a list of worker objects. A composite pattern was intended for the implementation, a group registration operation would have simply iterated over the list of workers and called registerWorker(). Via the composite, even non-cyclic nesting of worker groups would have been possible.

6.2 Improved Scheduling

The dynamic registration of workers, where they are not created by the main process but are added externally, naturally brings further challenges for such an implementation - the problem of external resources, which can differ greatly in their performance as well as optimizations. This becomes clear with an example: If two workers register with the main process, one of which is GPU- and the other CPU-supported, the conditions are completely different. In order to be able to manage this in the future, the Worker object already described in the previous subsection was further extended: Thus, a general type was also added to the WorkerID (initially as a state pattern, but since this does not change over the lifetime of a worker, a solution via an inheritance hierarchy would also be conceivable here, if not the preferred solution), as well as further information about expected performance, expected downtime and so on.

This feature is especially interesting from the software developer's point of view, because it allows extremely many possibilities for extensions (think here of automatic classification depending on the origin of the workers based on experience values!), but it also has a practical background: Depending on the type of worker, the main can distribute tasks or task packages optimally.

This feature is no longer visible in the final version - in the course of the development phase it was not considered further except for the impementation of the underlying class hierarchy.

6.3 Heartbeat and Resource Monitoring

In a distributed system, it is very easy for a part of the environment to fail - for example, because the hardware is damaged, a maintenance window occurs, there are network problems, and so on. To be prepared for such failures, a so-called heartbeat can help a lot - The concept here is that one part of a networker (in our case the worker) sends a signal to the main at regular intervals (about every 30 seconds) to show that it is still working. If one (or, depending on the implementationm several) heartbeats are missed, the main can react and reallocate tasks, change the load distribution or take other steps to continue providing functionality (in the best possible way).

This feature was developed in parallel with the registration or deregistration of resources and was an integral part of the prototype for a long time. It strongly favored the decision in favor of a state pattern for the workers, since the failure of a

corresponding worker would also have been represented using states - "pending" for a missed heartbeat and "disconnected" for the failure of three in a row. The objectification of work packages in the preparation phase was also intended as a preparation for this feature - For the redistribution of tasks where the processing worker has failed. So even if this feature was not implemented - due to careful planning at the beginning, the existing solution can be extended by this feature with comparatively few adjustments even after finalization with a completely different focus, with none of the original code of this feature remaining.

A possible extension, which however had no relevance for this project, is a so-called "handshake". In this case the handshake is based on reciprocity, not only the worker shows the main that he is still available, but the main sends similar signals to the worker. The worker thus receives the information that the task handed over by the main is still relevant and should be continued.

6.4 Random worker failures

Strictly speaking, this feature is not a real feature, but rather a mock[1] that helps to test the existing architecture. The basic idea here is that the provided, actually external workers are not provided externally, at least during the implementation period, but are provided by own resources. In order to be able to simulate reality as best as possible in this fairly closed - and thus reliable - environment, the workers should fail randomly with different probabilities under software control and no longer respond to the heartbeat.

7 Performance analysis

Our naive assessment of the possible performance of our algorithm on the problem size lead to a lower bound in computation time of ≈21 hours, see chapter 3.2.3. However, this assumed, that one iteration takes one clock cycle, which is probably far from realistic for a sequential solution using python. However, the goal of this practical was to parallelize the problem and as the problem is basically embarrassingly parallelizable faster run times should be possible.

7.1 Tools and Configurations

As stated before, the local computer was used to measure runtimes for the sequential approaches. The computer has a AMD Ryzen 5700g CPU, a M.2 SSD and 16 GB of Memory. All runs on the server were done using amp*** nodes which are Cascade Lake Intel Platinum 9242 machines[GWD]. They run with a clock speed of 2.3 GHz and provide 384 GB of Memory for 2x48 compute cores[GWD]. The scratch file system was used. Since a lot of data is produced by the algorithm, we decided not to store the data, because we did not want to waste resources and felt, that the storage solutions we

[1] In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways

used were far from optimal for the data. The input arguments for the scrips must be provided using the config.py file (number of PWMs, if files should be safed/pickled, etc.) and the parallel_run.sbatch file for resource specification (number of nodes, workers, memory, etc.).

7.2 Sequential Approach

This was the first prototype we produced. After some bug hunting and refactoring, we managed to run the script on the local machine on the smallest chromosome (chromosome 21) with two PWMs given as example PWMs by the bioinformatics workgroup. In our naivety, we hoped it would run for a few seconds, after all, this is a very small problem size. It took 3.95 Minutes to finish and we were a bit disappointed. After some more work, we benchmarked the sequential approach using one PWM and the whole Genome. This took ≈ 8 hours as shown in table 1. Is this fast or slow?
The program did around 1.87×10^{10} iterations (length genome times length PWM). 98.7% of the runtime was spend in the calculation phase, where we find the two for loops. The calculation phase took 28,340 s. This means, that the program ran at a speed of 659,845 iterations per second. The CPU runs at a base clock speed of 3.8 GHz, but can boost the clock speed to 3.8 GHz. When using the base clock speed, the CPU therefore needed 5,759 clock cycles on average per iteration. This seems very slow.

7.2.1 Sequential Approach using Numba

The version using Numba was much faster. However, we did not expect it to be that much faster. Our feeling was, that we would maybe get a 10x speedup, afterall we just added two lines of code. When it just took 8 minutes with the the calculation part just taking 2 minutes, we were flabbergasted. This is a speedup of 238 for the calculation part. Accordingly, the program ran at 157,142,857 iterations per second, again assuming it did 1.87×10^{10} iterations. Henceforth, the CPU needed 24 clock cycles on average to do one iteration. This sounds like a "to good to be true" performance in our opinion, as this includes calls to memory and divisions. There is probably some LLVM compiler magic happening and numba somehow managed to optimize the algorithm, but due to the limited time frame of this project we did not investigate further.
When running the same calculations on the server, which runs on 2.5 GHz and took 4:07 minutes, one gets 33 clock cycles per iteration. This validates the timing on the consumer hardware, as the consumer CPU probably ran on a higher clock speed than 3.8GHz. A simple rule of three gives 5.2 GHz which is higher than the CPU can provide. One explanation could be faster memory in the consumer PC or some sort of overhead on the server. Furthermore, to get really solid data, one should always run a benchmark like this multiple times which we did not do due to time reasons and a possible waste of resources in general.

7.3 Parallel Approach 1

The running times of the first approach are listed in Table 4. As this approach was just the step to approach 2, we just did three runs. With three measurements, we already see a speedup of ≈0.7 per worker. A good speedup behaviour was expected because of the high portion of not parallelizable program parts. This solution should in theory be embarassingly parallel when using many PWMs and a small granularity for the genome chunks. However, the performance improvement with this solution is limited by the main process, which will clog up with to many processes.

Since this approach was not about best performance, but about learning MPI, this will not be discussed further here. The much more interesting - and thus also analyzed in detail - approach is the second one, where the runtime is the focus of attention. This approach will be discussed in the next chapter.

7.4 Parallel Approach 2

Table 6: Runtime and speedup with increasing number of workers, calculating the KLD for the whole genome and 1 PWM.

Workers	Time	Time (s)	Speedup	Speedup/Worker
1	01:05:15	3915	1	1
8	00:10:15	615	6.36	0.79
10	00:09:00	540	5.91	0.59
16	00:08:24	504	7.76	0.48
21	00:06:59	419	9.34	0.44

Table 7: Runtime and speedup with increasing number of workers, calculating the KLD for the whole genome and 4 PWMs.

Workers	Time	Time (s)	Speedup	Speedup/Workers
1	03:32:02	12722	1	1
9	00:38:03	2283	5.57	0.61
12	00:26:45	1605	7.92	0.66
16	00:28:31	1711	7.43	0.46

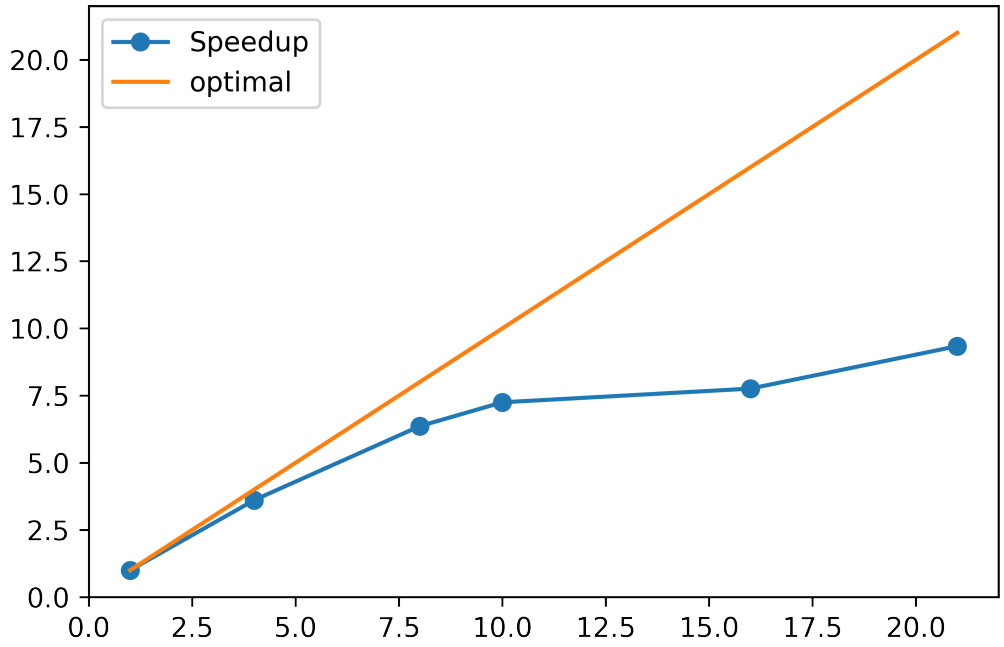

Figure 7: Speedup graph with one PWM. The actual speedup stays under the optimal, which is expected.

In tables 6 and 7 one can see the runtimes and speedups for one and four PWMs respectively. These values are problematic out of multiple reasons. Firstly, for the new approach to show its strength, one needs many PWMs, the more the better. Furthermore, each value has to be handled with care as it is not how strongly the values fluctuate. The first time we ran the four PWMs with one worker, it took 6:40:13 hours, the second time using the exact same parameters, it took 3:32:02 hours. The first value was clearly of, as we got better than optimal speedups and that is not possible. Nevertheless, more runs are required to get robust data.
One can see, that the speedup per worker is decreasing in both cases with increasing number of workers. This is probably because the calculations are not perfectly parallelizable given our course granularity of the genome sequence chunks. In the end, some workers are still calculating on the larger chromosomes, while others are idleing. This behaviour can of course also be observed in the speedup diagrams.
When looking at the times, one wonders why the parallel approach using just one process takes so long compared with the sequential approach (1 hour vs 14 minutes). We do not know the exact source of this difference, as we did not manage to get a profiler running on the server in the given timeframe. However, one possible reason is, that the worker process pickles every chromosome respecting our specification, see Figure 6.
As expected, in Figure 9 we can see, that the calculation with 4 PWMs is more efficient than the calculations with one PWM.

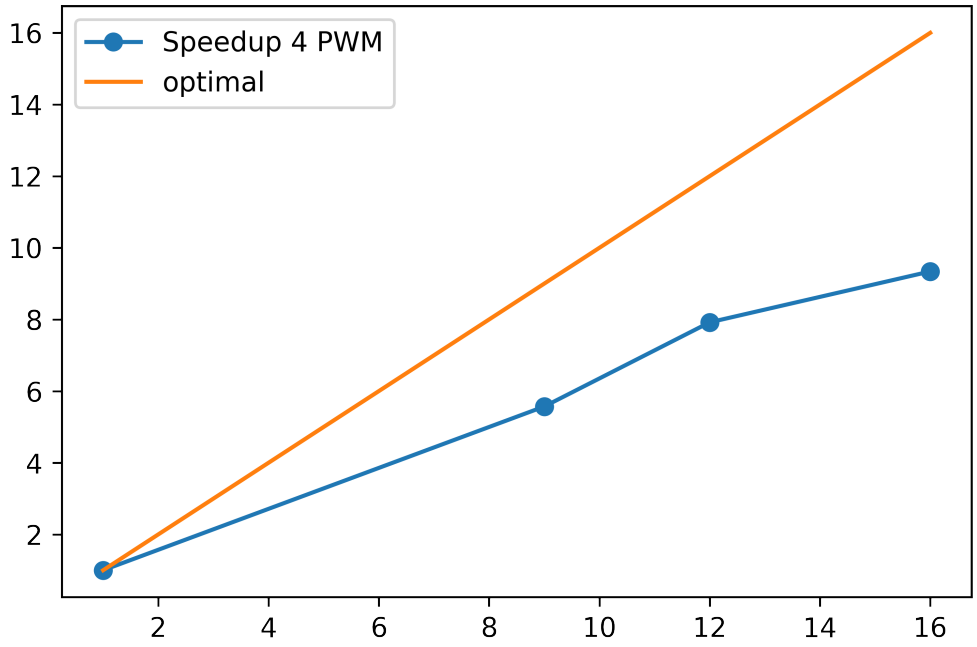

Figure 8: Speedup graph with four PWMs. As with the single PWM pass, the speedup remains below the linear optimal speedup, and the curve flattens out as more workers are involved in the operation.

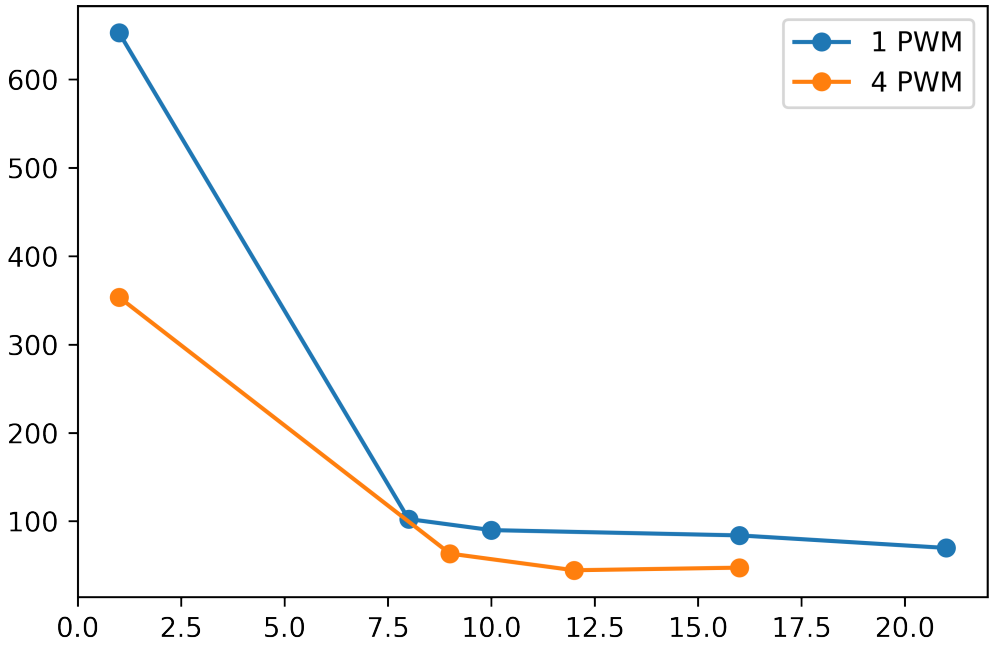

Figure 9: Combined Speedup graph with one and four PWMs.

8 Conclusion

At the end of such a report, it is important to look back on the work that has been done, to look at what has been achieved and learned, and to draw a conclusion. This is exactly what this final chapter of the report on the Practical Project in High Performance Computing is about.

8.1 Summary of the project

This report describes the implementation of several approaches to perform a simple but complex mathematical computation on a human genome. This involved first laying some groundwork - biological, mathematical, and informational. Then, the problem at hand was analyzed in detail and solution strategies were developed - first in a technically ideal world, then in the real one with all the given technical limitations. Prior to implementation, expectations regarding performance could be formulated based on these.

A total of 3 solutions were implemented and examined in more detail, one sequential and two parallel approaches. These were each tested under different circumstances and the results analyzed in more detail. Subsequently, the insights gained were compared with the hypotheses so that an overall evaluation was possible.

In the end, a model solution for the problem at hand could be handed over to the project's commissioners, a working group of the bioinformatics faculty. On the way there, however, different challenges had to be overcome, which will be discussed in more detail in the following section.

8.2 Project review

A few goals were defined in chapter 3.2.4.

Firstly, the top priority goal was to create a high-performance solution for calculating the Kullback-Leibler divergence. This goal was achieved, the processing time could be reduced significantly. The goal to achieve this in less than one hour was not reached, but this does not diminish the overall success.

The second goal was to develop the solution in such a way that a main process manages many subprocesses and provides various functionalities for this purpose. This goal was achieved to a limited extent; although it was possible to develop such an infrastructure on the whole, significantly fewer management options were implemented than originally planned. For detailed information about the reasons behind that, see chapter 6.

The third and most important goal is beyond software functionalities - as a project in an educational institution, learning success comes first. The knowledge gain was extremely high, in the technical area (such as the use of MPI, numba or design patterns), but also in how such projects are approached in general. Moreover, the solution was handed over to the bioinformatics work group, who are extremely satisfied with the result and can use it as a basis for future projects. We will stay in touch with the working group. The knowledge that this project did not fulfill a pure end in itself compensates for the possibly unfavorable choice of project. A completely self-selected topic without external influences could possibly have led to better overall results with regard to the performance analysis.

It should be noted, that the biggest challenges in this project were the smallest details. Particularly during implementation, there were always small-seeming obstacles and bugs, which, however, extremely prolonged the overall processing time - In many cases, it was necessary to shimmy from one error message to the next during debugging, which required an enormous amount of nerve. Pair programming, i.e. working together on one computer, provided a remedy. This eased the debugging and could contribute to a higher code quality, however, the work could hardly be parallelized - so that the testing and benchmarking at the end of the implementation time unfortunately did not get as much attention as originally planned.

It is still difficult to say whether more parallel work would have led to qualitatively and quantitatively better results - because, besides the code that could be handed over to the bioinformatics work group in good quality, there is also the certainty that a lot was learned. This aspect will be discussed in more detail in the following subchapter.

8.3 Learning achievements and possible improvements

The skills learned in this project can be roughly divided into two categories: Project management and programming, i.e. on the one hand the organizational part of the project, and on the other hand the technical part.

8.3.1 Project management

Implementing a solution to a fairly simple problem like this one can be much more involved than one might expect at the outset - this project has made that clear. The requirement to implement such a computation in such a way that a result with good performance is achieved, while at the same time having a code base that can be used as illustrative material for a faculty allows for a lot of leeway as far as the implementation is concerned. At this point, better planning would have been necessary at the beginning to save a lot of unnecessary work for the overall result, the chapter on removed features shows the urgency for improvement here. The approach of implementing as much as possible together was a success in retrospect - the pair programming itself can be improved in the future, however, this was expected as it ran without any major plan or preparation. Techniques such as mob programming, where developers take turns at fixed times and dictate who is not at the keyboard, can contribute to increased effectiveness in the future - and thus automatically to better time management. Basically, the realization remains that even for apparently smaller projects, significantly more time must be planned - or it must be better used.

8.3.2 Programming

In terms of the technical learning curve, the project was a complete success. On the one hand, the topic itself required new skills for both of us, such as the general mindset behind parallel programming, but also direct skills such as programming with numba and MPI. In addition, we were able to use the project to learn a lot from each other - With very different backgrounds (a Python-centric, academic background for Tim van den Berg on the one hand, and an industrial developer background for Vincenz Dumann), so much knowledge was transferred. For example, the effective use of design patterns, or the exploitation of Python's strengths (and of course the avoidance of its weaknesses). Especially the performance increase through the use of numba will remain in memory.

References

[bro22] broadInstitute. *BroadInstitute*. https://www.broadinstitute.org/blog/pages-first-human-genome. Accessed: 16.09.2022. 2022.

[Cam+16] Neil A Campbell et al. *Campbell biologie*. Pearson, 2016.

[Gam97] Erich Gamma. *Design Patterns. Elements of Reusable Object-Oriented Software*. Prentice Hall, 1997.

[GWD] GWDG. *High performance computing*. URL: https://docs.gwdg.de/doku.php?id=en%3Aservices%3Aapplication_services%3Ahigh_performance_computing%3Astart.

[JA+] astro-Mondragon JA et al. *About JASPAR*. URL: https://jaspar.genereg.net/about/.

[Jüs] Prof. Thomas Jüstel. *Desoxyribonukleinsäure (engl.: deoxyribonucleic acid, DNA)*. URL: https://www.fh-muenster.de/ciw/downloads/personal/juestel/juestel/DNA.pdf.

[NCf13] NCfBI. *National Center for Biotechnology Information*. 2013. URL: https://www.ncbi.nlm.nih.gov/genome/guide/human/.

[num22] numba. *Numba - official Documentation: Enum*. https://numba.pydata.org/. Accessed: 16.09.2022. 2022.

[pyt22] pythondoc. *Python - official Documentation: Enum*. https://docs.python.org/3/library/enum.html. Accessed: 16.09.2022. 2022.

[S K51] R. A. Leibler S. Kullback. *Design Patterns. Elements of Reusable Object-Oriented Software*. ProjectEuclid, 1951.

[wik22] wikimedia. *Human Chromosome*. https://upload.wikimedia.org/wikipedia/commons/b/b2/Karyotype.png. Accessed: 16.09.2022. 2022.

A Work sharing

Most of the implementation was done in pair programming, some smaller parts were splitted.

A.1 Tim van den Berg

In addition to the pair programming, Tim van den Berg was mainly doing the biological parts and the DevOps: He acquired the topic and was in charge of making the project run on the server.

In addition, Tim brought a lot of python knowledge to the project and can be seen as main developer.

A.2 Vincenz Dumann

In addition to the pair programming, Vincenz Dumann was mainly responsible for the presentation and the writing of the report. Regarding the technical part, he mainly brought the knowledge about design strategies into the team and was in charge of refactoring and overhauling the code.

B Source Code

The complete source code can be found on the Community Edition GitLab instance: `https://gitlab-ce.gwdg.de/tim.vandenberg/practical_hpc`